Seminar 6

1 default, delete, override & final

Incepand cu C++ 11 au fost adaugate cateva noi cuvinte cheie pentru a putea scrie cod mai expresiv si a putea da implementa cat mai bine functionalitatile claselor noastre.

default

Atunci cand implementam constructorul cu parameterii/de copiere pierde constructorul implicit (cel fara parametrii) oferit de compilator. De cele mai multe ori, acest constructor este mai mult decat necesar pentru o clasa iar implementarea unui constructor ia timp si ocupa spatiu in fisierele sursa. Cuvantul cheie `default` ne permite sa ii spunem compilatorului ca noi vrem in continuare sa genereze construtorul implicit. Prin atribuirea explicita unei signaturi din clasa a cuvantului cheie `default` putem sa avem in continuare parti ale clasei noastre definite de compilator:

```
    [<tip - retur>] <nume> (<lista - parametrii>) = default;
```

Exemplu:

```
class C {
public:
    C () = default;  // avem in continuare constructorul fara parametrii
    C (const C& c) {cout << "cpy-C";}
    C& operator= (const C&) = default;
};
```

Cuvantul cheie `default` poate fi folosit si pentru mentionarea explicita in cod ca pentru clasa noastra vom avea operatori/componente care sunt definite de catre compilator.

delete

Sunt situati in care, pentru anumite clase, nu dorim sa avem anumite functionalitati care sunt definite automat de compilator. Spre exemplu, ne dorim ca obiectele unei clase sa fie folosite in apeluri prin valoare (sau mai general sa se faca copii). Pentru a realiza asta, trebuie sa ii spunem compilatorului ca nu vrem sa ne funizeze implementarea pentru constructorul de copiere si operatorul de atribuire. Acest tip de functionalitate se realizeaza prin folosirea cuvantului cheie `delete`, intr-o maniera similara cu folosirea cuvantului cheie `default`:

```
    [<tip - retur>] <nume> (<lista - parametrii>) = delete;
```

Exemplu:

```
class C {
public:
    C (const C& c) = delete;
    C& operator= (const C&) = delete;
    // folosirea constructorului de copiere sau a operatorului de atribuire
    // va resulta intr-o eroare de compilare
};
```

Tot prin folosirea cuvantului cheie `delete` mai putem sa renuntam la anumite metode mostenite daca functionalitatile oferite nu sunt necesare/potrivite pentru clasa derivata:

```cpp
class C {
public:
    void foo ();
};

class D : public C {
public:
    void foo () = delete;
    // nu mai poate fi apelata metoda foo folsing obiecte te tip D.
}
```

Avem totusi si cateva restrictii cand vine vorba de stergerea metodelor mostenite:

- nu putem sterge metode virtuale;
- se poate apela totusi metoda din baza folosing `ob.Base::method();` ;

Mai putem folosi `delete` pentru nu permite anumite liste de apeluri prin ale unei functii polimorphice sau anumite instantiri de deplate-uri:

```cpp
template <typename T> T add (T a, T b) {
    return a+b;
}
template <> int add<int> (int a, int b) = delete;

void polymorphic (long);
void polymorphic (int) = delete;

int main () {
    polymorphic(2l); // apelul cu type long e permis
    polymorphic(2);  // eroare, nu e permis apelul cu int
    add(2.3f, 2.5f); // instanierea cu float este permisa
    add<int> (2, 4); // nu e permisa instantierea implicita/explicita cu int
}
```

override

Cunvantul cheie `override` ne permite sa marcam explicit in code suprascrierea unei metode virtuale. Avantajul folosirii `override` este ca acesta va genera erori de compilare daca signatura marcata pentru suprascriere nu este virtuala in baza sau nu e identica cu cea din baza, i.e. cod mai sigur:

```
    [<tip - retur>] <nume> (<lista - parametrii>) [<specifier> ...] override;
```

Exemplu:

```cpp
class B {
public:
    virtual void foo ();
    void bar();
};

class D : public B {
public:
    void foo () override;  // este permis, foo e virtual
    void foo () const override; // nu este permis, signatura e gresita
    void bar () override; // nu este permis, bar nu e virtual
    void baz () override; // nu e permis, baz nu exista in baza
};
```

final

Cuvantul cheie `final` ne ajuta sa oprim orice alta suprascriere a unei metode virtuale in clasa derivata sau sa oprim orice viitoare mostenire a unei clase. Sintaxa:

```
    // metoda final
    [<tip-retur>] <nume> (<lista-parametrii>) [<specifier> ...] final;
    // clasa final
    class <tip-retur> final [:<spec-access1> <Base1>, ...] {
    };
```

```
class B {
public:
    virtual void foo ();
};

class D: public B {
public:
    // foo este final in D, orice suprascriere in derivata
    // va rezulta in eroare de compilare
    void foo () override final;
};

// clasa E este final, nu mai poate fi mostenita
class E final: public D {
public:
    // eroare, foo nu poate fi suprascris
    void foo () override;
};

// eroare E nu poate fi mostenit
class F: public E {
};
```

Restrictii similare cu `override` avem si pentru `final` : nu putem folosi acest specificator pentru metode care nu sunt virtuale.

2 initializer_list

Intotdeauna ne-am pus problema cum putem avea un constructor cu care sa putem crea un vector (orice clasa de tip colectie) cu elemente in el sau sa declar pe moment obiecte temporare pentru clasele noastre fara sa fie nevoie sa apelam constructori. Raspunsul este dat de lista de initializare.

Lista de initializare este un concept nou adauga in C++ 11 care ne permite sa declaram obiecte si sa le initializa direct cu valori fara sa fie nevoie sa apelam explicit un constructor. Acest lucru este posibil pentru ca in momentul folosirii unei liste `{a, b, ...}` (similar cu declarea statica a unui array) compilatorul va cauta construtorul in care poate folosi primul element din lista ca prim parametru la constructor, al doilea element din lista ca al doilea parametru la constructor si asa mai departe. Mai mult decat atat, putem sa furnizam ca parametrii liste de initializare catre functii care accepta un obiect de tipul clasei noastre.
Exemplu:

```
class A {
    int x, y;
public:
    A(int i, int j) : x(i), y(j) {}
};

class B {
    A a;
    float f;
public:
    B (float n, A ob) : a(ob), f(n) {}
};
```

```
13
14 class C {
15     int x;
16 public:
17     C(int i) : x(i){}
18 };
19
20 void foo(A);
21
22 int main () {
23     A a1 = {3, 10}, a2{4, 0};
24     B b = {2.3f, {3, 12}};
25     C c = 3;
26     foo({22, 55});
27 }
```

Toate aceaste sunt posibile deoarece compilatorul pot realiza conversii implicite pentru clasele noatre folosinduse de constructor. In exemplul de mai sus se poate observa ca putem asigna un intreg catre un obiect de tip `C` . Acest comportament este posibil doar la operatiile de atribuire la declarare. Pentru tipuri de date mai complexe putem sa declaram liste imbricate atat timp cat fiecare list pe care o adaugam noi se mapeaza pe un constructor.

Pentru structuri in care numarul de elemente din lista de initializare este variabil, putem folosi constructorul lista de initializare:

```
1 <nume - clasa> (initializer_list<T>);
```

Acest constructor primeste un parameteru de tip `initializer_list<T>` , definit in headerul `initializer_list` , care reprezinta o list de valori de dimensiune n cu care poate fi initializat obiectul nostru.
Exemplu:

```
1 #include <iostream>
2 using namespace std;
3
4 class C {
5     int s;
6 public:
7     C (initializer_list<int> l) : s(0) {
8         for (int i : l) { s += i;}
9     }
10    friend ostream& operator << (ostream& out, const C& c) {
11        out << c.s; return out;
12    }
13 };
14
15 int main () {
16     C c {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
17     cout << c << endl;
18     return 0;
19 }
```

3 Expresii lambda

Programarea functionala este o noua paradigma de programare in care functiile sunt tratate ca orice alt tip de date. Astfel, pentru a introduce aceasta paradigma, in C++ 11 au fost adaugate expressii lamda.

Expressile lambda ne permit sa definim functii anonime pe care sa le pasam ca parameterii catre alte functii (e.g. sort, find). Sintaxa pentru a defini o expresie lamda este urmatoare:

```
1 [<capturi>] (<lista - parametrii>) -> <tip - retur> {
2     /* corp expresie */
3 }
```

Specificarea tipului de retur nu este necesara daca acesta poate fi dedus de compilator din corpul expresiei.

```
[] (int i) { // e ok, compilatorul deduce ca se intoarce un intreg.
    return i + 3;
}

[] (int i) -> double {  // trebuie sa ii spunem compilatorului ca tipul introdus
    if (i % 2 == 1) {   // deoarece el nu poate alege intre int si double
        return i / 2.0;
    } else {
        return i;
    }
}
```

Pentru a putea folosi variabile definite in afara expresiei lambda trebuie sa "capturam acea variabila" in lambda expresie. Asta se poate realiza in mai multe feluri:

- `[=]` – captureaza prin valoare toate variabilele externe expresie lamda
- `[&]` – captureaza prin referinta toate variabilele externe expresiei lambda
- `[&x, y]` – captureaza variabila `x` prin referinta si variabila `y` prin valoare

```
#include <iostream>
#include <vector>
using namespace std;

int main () {
    vector<int> v;
    for (int i = 1; i < 20; i ++) {
        v.push_back(i);
    }
    for (int i : v) {
        cout << i << " ";
    }
    cout << endl;
    sort(v.begin(), v.end(), [](int i, int j) {return i % 2;});
    for (int i : v) {
        cout << i << " ";
    }
    cout << endl;
}
```

```
#include <iostream>
using namespace std;

int main () {
    int i = 5, j;
    cin >> j;
    auto is_bigger = [i] (const int x) {
        return x > i;
    };
    is_bigger(j);
    return 0
}
```

```
#include <iostream>
using namespace std;

int main () {
    int i = 5, j;
    cin >> j;
    auto is_bigger = [&i] (const int x) {
        if (x > i) {
            i = x;
        } else {
```

```
            i--;
        }
    };
    is_bigger(j);
    return 0
}
```

4 Move semnatics

Pentru structuri de date mai complexe unde crearea copiilor este foarte costisitoare, genul acesta de apel va consuma foarte mult timp doar pentru a executa alocari si dezalocari de memorie. ,

```
#include <iostream>
using namespace std;

class C {
    int i;
    static int count;
public:
    C () {++count; i = count; cout << "C" << i << endl;};
    C(const C& c) : i(c.i) {cout << "C-cpy" << c.i << endl;}
    C& operator= (const C& c) {cout << "C&=" << c.i << " discarding " << i << endl; i
     = c.i; return *this;}
    friend C operator+ (C x, C y) {
        return C();
    }
    ~C() {cout<<"~C" << i <<endl;}
};

int C::count = 0;

int main () {
    C c1, c2;
    c2 = c1 + C();
}

// C1
// C2
// C-cpy1
// C3
// C4
// C&=4 discarding 2
// ~C4
// ~C3
// ~C1
// ~C4
// ~C1
```

Astfel a fost introdusa semnatica de mutare prin care, in loc sa se fac o copie, se muta cu totul starea obiectului.
Constructorul de mutare este un nou tip de constructor introdus in C++11 care, al carui scop este sa mute starea obiectului intr-un obiect nou, sarind astfel peste copiere bucatica cu bucatica a obiectului pasat ca parametru. Sintaxa:

```
    <nume - clasa> (<nume - clasa>&&);
```

Acest tip de constructor este definit automat de compilator doar in anumite situatii

La pachet cu constructorul de mutare avem si operatorul de atribuire prin mutare (care este definit implicit de compilator doar in anumite situatii):

```
    <nume - clasa>& operator=(<nume - clasa>&&);
```

Pentru implementa corect mutarea, este recomandat sa folositi functia `move` care este implementata pentru toate tipurile de date primitive. Daca ne dorim ca functia `move` sa functioneze pentru tipurile

noastre de date corect, trebuie sa ne asiguram ca am implementat constructorul mutare si operatorul de atribuire prin mutare:

```
T p;
T q = move(p);
```

Cum se modifica codul?

```
#include <iostream>
using namespace std;

class C {
    int i;
    static int count;
public:
    C () {++count; i = count; cout << "C" << i << endl;};
    C(C& c) : i(c.i) {cout << "C-cpy" << c.i << endl;}
    C(C&& c) : i(c.i) {cout << "C-move" << c.i << endl; c.i = 0;}
    C& operator= (C& c) {cout << "C&=" << c.i << endl; i = c.i; return *this;}
    C& operator= (C&& c) {cout <<"C&&=" << c.i << endl; i = c.i; c.i = 0; return *
    this;}
    C operator+ (const C&) {
        return C();
    }
    ~C() {cout<<"~C" << i <<endl;}
};

int C::count = 0;

int main () {
    C c1, c2, c3;
    c3 = c1 + c2;
}

// C1
// C2
// C3
// C4
// C&&=4
// ~C0
// ~C4
// ~C2
// ~C1
```

De ce afisarile nu sunt chiar asteptate? In versiunile mai noi de compilatoare, in compilatorul va deduce daca e nevoie sa apeleze constructori sa copieze/mute, si daca nu e necesar, va prelungi perioada de viata a anumitor obiecte pentru a nu executa apeluri catre constructori (C++14).

```
#include <iostream>
using namespace std;

class C {
public:
    C() { cout << "C" << endl; }
    ~C() { cout << "~C" << endl; }
    C(const C& c) { cout << "C-cpy" << endl; }
    C& operator=(C c) { cout << "C=" << endl; return *this;}
};

C foo () {
    return C();
}


int main () {
    C c1 = C();
    C c3 = foo();
}

```

```
22 //  C
23 //  C
24 //  ~C
25 //  ~C
```